Freescale Semiconductor
Application Note

AN3506
Rev. 0, 12/2007

Migration from TI MSP430 to 9S08QE128 or MCF51QE128 Flexis Microcontrollers

Enhancing Low-Power Performance

by: Inga Harris
8-bit Microcontroller Applications Engineer
East Kilbride, Scotland

1 Introduction

From the RS08 to our highest-performance ColdFire® V4 devices, the Controller Continuum provides compatibility for an easy migration path up or down the performance spectrum. The connection point on the Controller Continuum is where complimentary families of S08 and ColdFire V1 microcontrollers share a common set of peripherals and development tools to deliver the ultimate in migration flexibility. Pin-for-pin compatibility between many devices allows controller exchanges without redesigning the board. Development tool compaitibility enables seamless porting with one simple recompilation step. The MC9S08QE128 and the MCF51QE128 are the first products in the Flexis series.

Flexis is a single development tool that eases migration between 8-bit (S08) and 32-bit (CFV1). Also, it is a common peripheral set to preserve software investment between 8-bit and 32-bit and pin compatibility wherever practical to maximize hardware reuse when moving between 8-bit and 32-bit.

Contents

freescale™ semiconductor

This application note shows how to convert a typical MSP430FG4619 application into a QE128 application. The software conversion was written using IAR Embedded WorkBench Kickstart for MSP430 V3 and Freescale CodeWarrior for Microcontrollers V6.0.

The hardware tool used to develop the application code is the MSP430 USB-Debug-Interface MSP-FET430UIF with a custom target board for the MSP430FG4619 as shown in Figure 1. The QE128 hardware tool used is the SofTec Microsystems EVBQE128 Starter Kit, shown in Figure 2, can operate with the MC9S08QE128 or the MCF51QE128 in 80-Pin package. The image sizes are approximately relative.

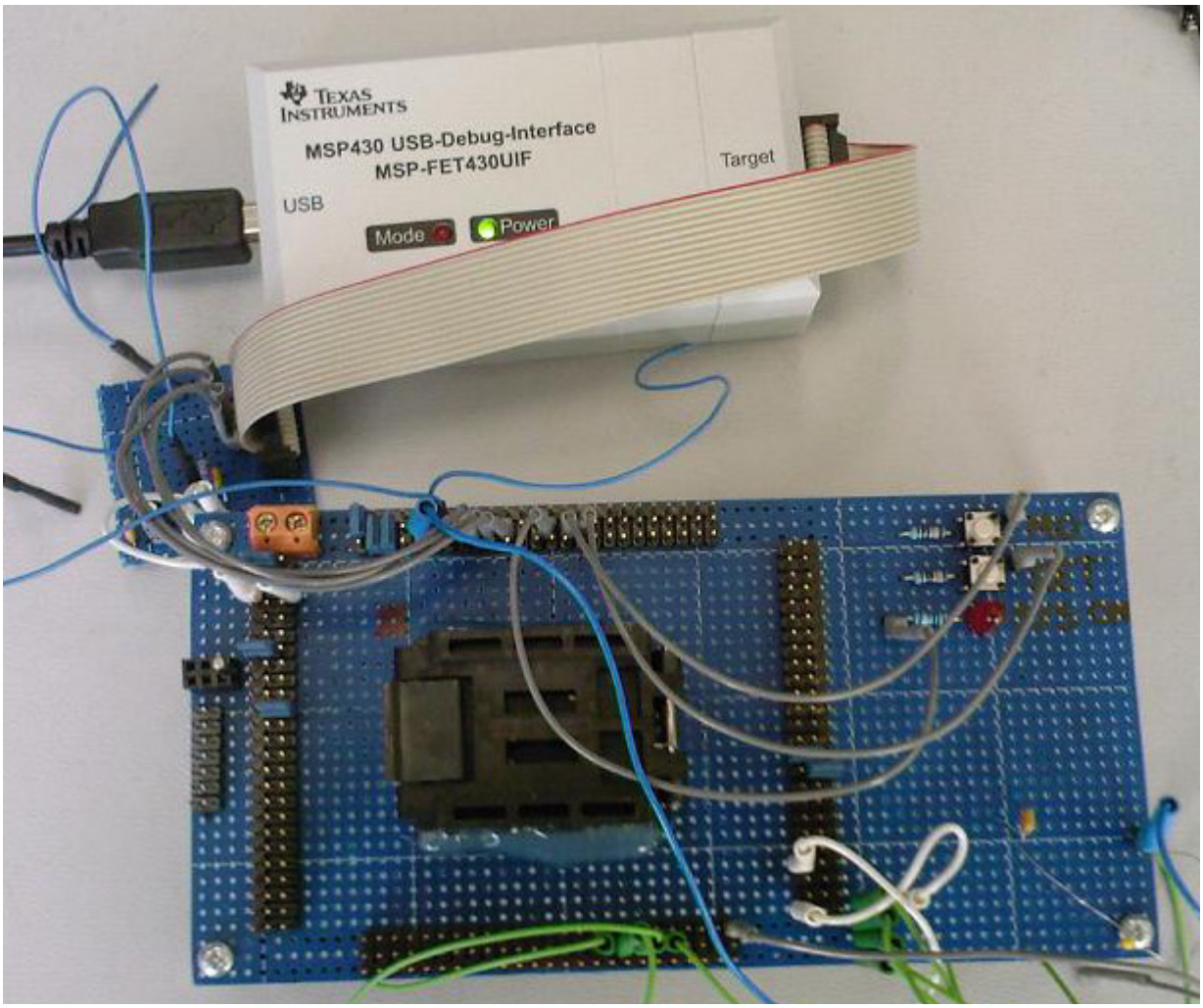

Figure 1. Custom MSP430FG4619 board using MSP-FET430UIF

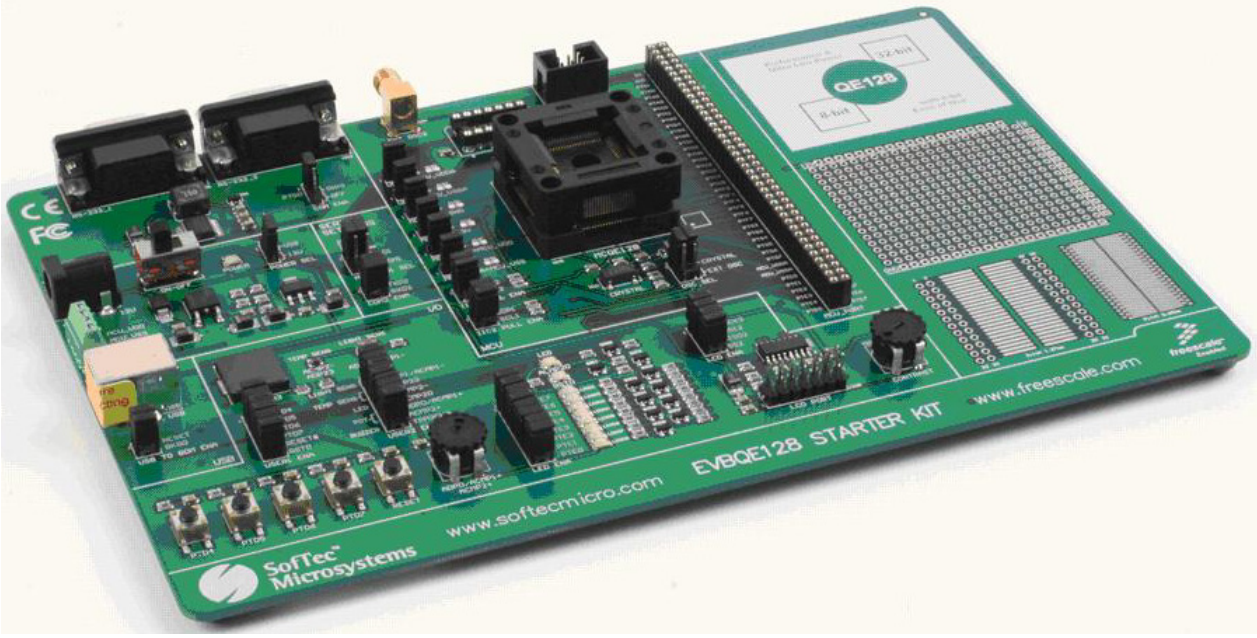

Figure 2. EVBQE128 Starter Kit

NOTE

When QE128 is referenced in this document, this statement is applicable to MC9S08QE128 and MCF51QE128. If the statement is only applicable to one of the devices, the full part number is used.

NOTE

The details on the MSP430 are based on publicly available data and form the basis of a comparative analysis to the QE128 devices on a number of features. For an in depth analysis of the MSP430 itself, Texas Instruments (TI) should be contacted directly.

2 Application Example

2.1 Ultra Low Power Thermostat with 12-bit A/D Conversion

The block diagram shown in Figure 3 shows the application hardware configuration for the MSP430FG4619 and the QE128 thermostat system. The application features the real time clock, 12-bit analog-to-digital convertor, SPI, GPIO, timer, and pushbuttons for state and setpoint inputs. The main loop for this implementation runs once every second, prompted by an interrupt from the RTC module. Between interrupts, the device is in Stop3/LPM3 resulting in low power operation.

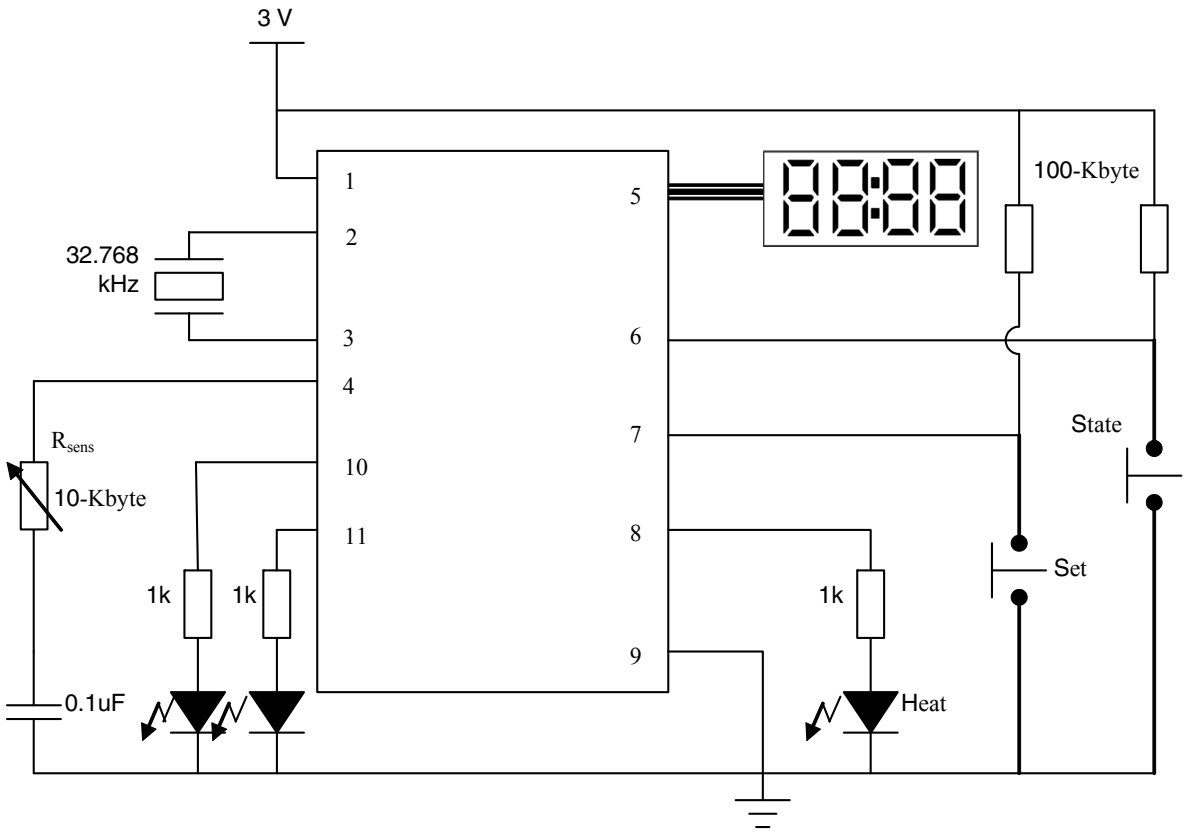

Figure 3. Thermostat Block Diagram

The pins in the above block diagram have been numbered 1 to 11 because the actual pin functions and labels for the two devices are detailed in Table 1. Pins 10 and 11 are optional and only needed to aid debugging because you can see on the board what status the code is currently in.

Table 1. Thermostat Pin Function and Assignment

Pin No.	Application Function	MSP430FG4619 Pin	QE128 Pin
1	3V Supply	VCC	VDD
2	External Crystal	XIN	XTAL
3	External Crystal	XOUT	EXTAL
4	AD Input	AD0/P6.0	PTA0/ADP0
5	LCD Control	USCI_A0 pins	SPI2 pins
6	GPIO	P1.1	PTD4
7	GPIO	P1.2	PTD5
8	GPIO	P1.0	PTE0
9	Ground	VSS	VSS
10	Mode Indicator LED	P1.7	PTE6
11	Mode Indicator LED	P1.8	PTE7

2.1.1 Design Description

The RTC interrupt flag is set every second and the clock function is executed imediately to ensure it is always done and no seconds are lost. To minimise time spent in the main software loop, the maximum bus frequency possible with the 32 kHz crystal is used. Because the application relies on seconds and not minutes, which means the shortest interrupt period is one minute, the real time clock function available on the MSP430FG4619 is not be used .

The ADC takes a voltage reading from the application's thermistor (Rtherm) every 30 seconds and compares it to a target value set by the user. The CPU then decides if the room temperature needs adjusting, indicated by the heat LED.

The SPI transmits three parameters (one at a time): current room temperature, temperature setpoint and time, controlled by the state variable.

The target temperature is adjusted by the user using the set button while in set state. When the device is in the set state, pressing the set button repeatedly cycles around the temperature range 10 to 40 degrees celcius. After the correct temperature is reached, the mode is exited via the state button.

The time is updated by the set button when the device is in the time state. Each button press adds one minute to the time (24-hour clock). Time state is only enabled for 10 seconds and automatically moves to temp state.

Because the MSP430FG4619 has an LCD driver on chip, this module could control the LCD display directly. However, for a fair power comparison between the two devices in this example, the SPI is used to send out the data on the QE128 and the MSP430FG4619. Where the MSP430FG4619 code uses USCI_A0, the QE128 version of the application uses SPI2 to send out the equivalent messages to an SPI enabled peripheral.

NOTE

Many applications no longer use on chip LCD modules because the ribbon cable connecting the chip to the display can break and show the wrong information (missing segments) on the screen. If a serial comm link is used, cable breakages cause the whole screen to go blank, which indicates an issue. An external I2C enabled, 128 segment LCD controller, such as the NXP PCF8562, can be used with the QE128 devices for a display solution.

The CPU code flow is almost identical for the MSP430FG4619, the MC9S08QE128, and MCF51QE128 devices, with only peripheral code changes between the devices.

2.1.2 Application Flow Charts

The application code has two sections: the Main loop (including subroutines) and the Interrupt Service Routines (ISRs). In this thermostat implementation, the MCU wakes up every second by the RTC interrupt from Stop3/LPM3 mode to:

1. Check for a button press, and attend to is necessary
2. Adjust the state if necessary,
3. Take a new temp reading (if 30 seconds passed),

4. Update the display - always

2.1.2.1 Main Loop

Figure 4 shows the main program flow chart, including all sub routines.

NOTE

Two buttons should not be pressed at the same time because only one is active per cycle and the STATE button has precedence.

The routine always checks that the cycle flag has been set, which it should be during error free execution. Depending on the mode, follow a path through to the display function.

The MCU then enters a low-power mode: Stop3 or LPM3 depending on the MCU.

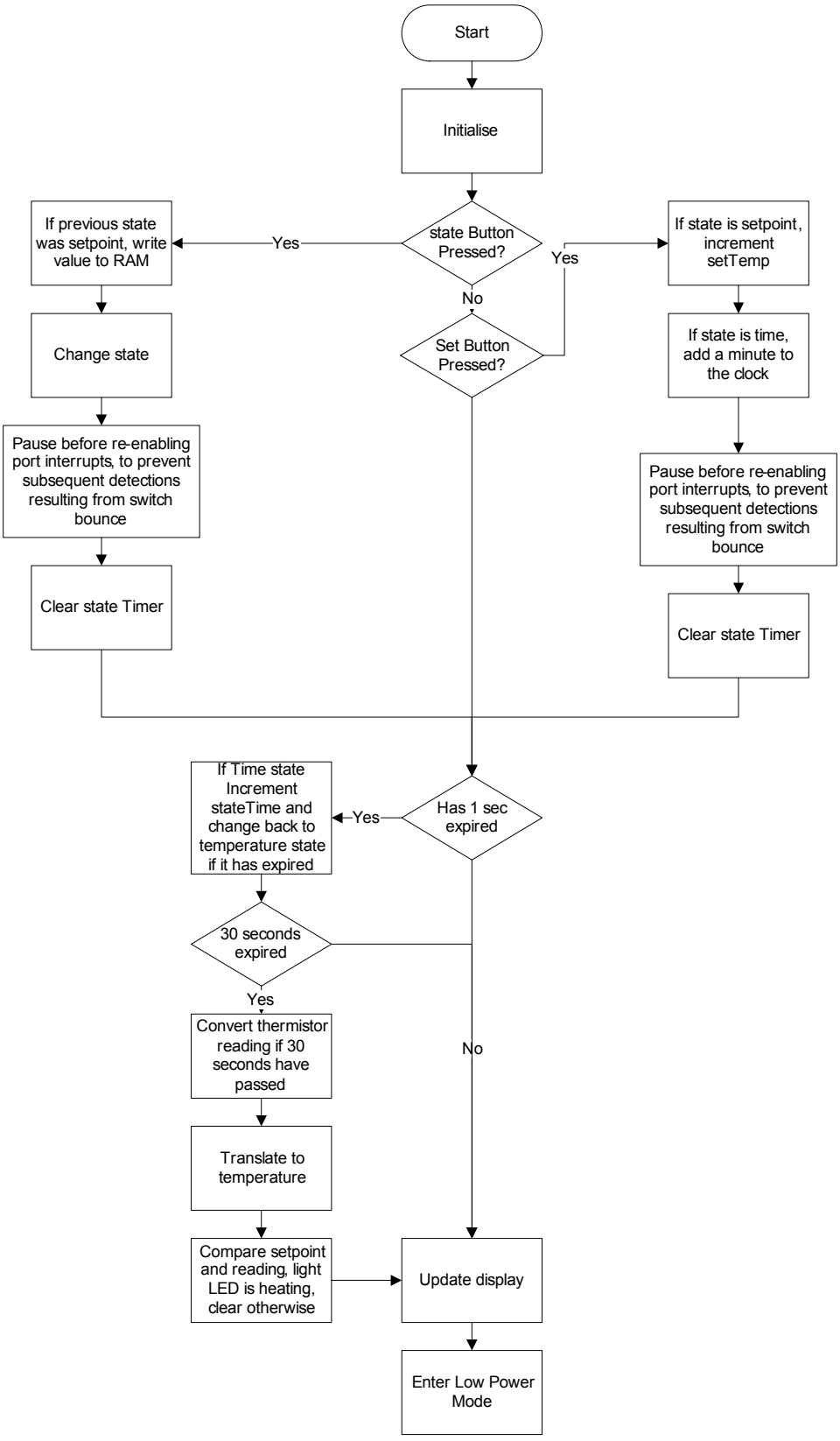

Figure 4. Main Program Flow Chart (including sub-functions)

2.1.2.2 Interrupt Service Routines

Figure 5 shows the three interrupt service routines (ISRs) flow charts.

The debounce routine clears the timer overflow flag and takes it back to the run mode.

The real time clock routine manages the clock functions, sets the SecCycleFlag, and stops a KBI from disturbing the program flow back to main.

The SPI rouitine clears any flags that may be set as errors and received messages are not supported in this transmission only application.

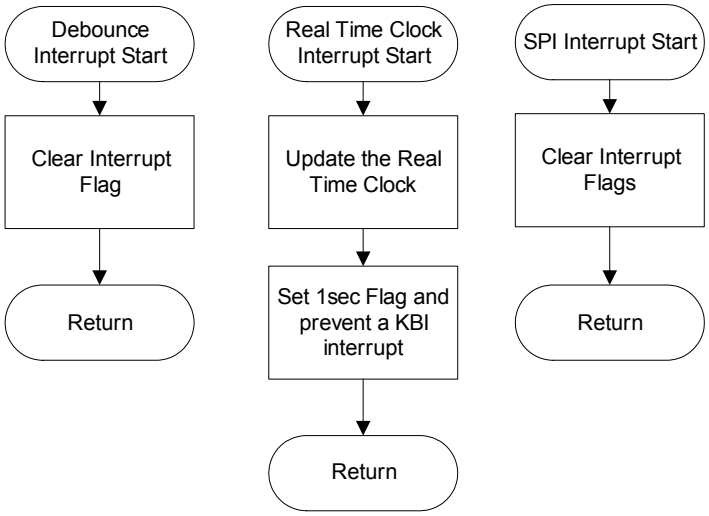

Figure 5. Interrupt Service Routine Flow Charts

3 Design Conversion

3.1 Hardware

Freescale Semiconductor and Texas Instruments devices operate at the same voltage. Because of this, the only hardware changes required are adjusting the pin out, as shown in Table 1, and swapping hardware debugging tools.

3.2 Software

The easiest way to transfer a project from one device to another, when the tool suites are not compatible, is to create a blank template in the new environment and start copying in defines and functions, starting with main, and following the flow chart paths. Inevitably, some conversions of register names are erroneous and following the flow chart enables easier debugging. Some functions require major reworking because the modules used operate differently. Historically, differences between the S08, Coldfire V1 devices, and linkers means that vector tables and ISRs should be references in a specific, compatible way, as described in section 7 of AN3465, Migrating within the Controller Continuum.

NOTE

The following proceedure is one of many ways to convert a project and preferences vary between engineers depend on tool capailities.

3.2.1 Project Clean-Up

The key to a successful transfer is to start with a project that has redundant code removed and a solid understanding of what the code and modules do in the application. Order and timing knowledge is also an advantage, especially if the projects success is to be measured in real terms, e.g. power consumption savings.

Unfortunately, many inheritted programs are not well documented, but it is worthwhile to spend time to understand what the code does and add comments to help conversion to the other device for future modifications.

After the application and code is understood and any code improvements have been made, the conversion can start.

3.2.2 Create New Project using CodeWarrior for Microcontrollers V6.0

Start Codewarrior for Microcontrollers V6.0 by double clicking on the desktop icon or the program in the program list.

If Start-Up Dialog Box does not appear, you can go to the File menu and open the dialog box or go straight to New Project.

Click on Create New Project.

Figure 6. CodeWarrior Start-Up Dialog Box

Next, you must select the MCU derivative and the connection type then appears. This application note uses the DEMOQE128 board from Softec. Click Next.

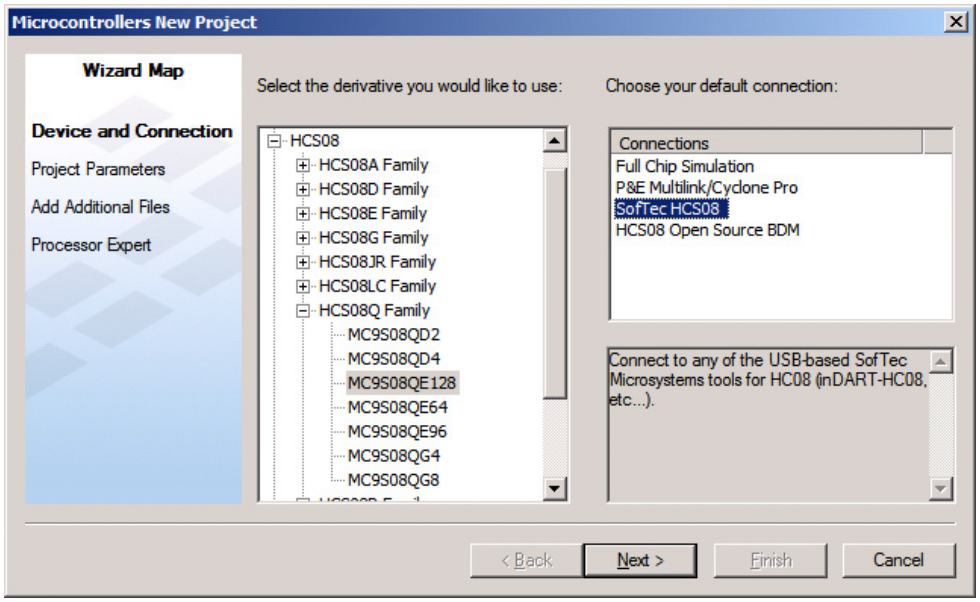

Figure 7. Device And Connection Type

The following screen is the final step required to create the project. The Project Wizard prompts for you to select the language create the project with the given name and location. Click Finish.

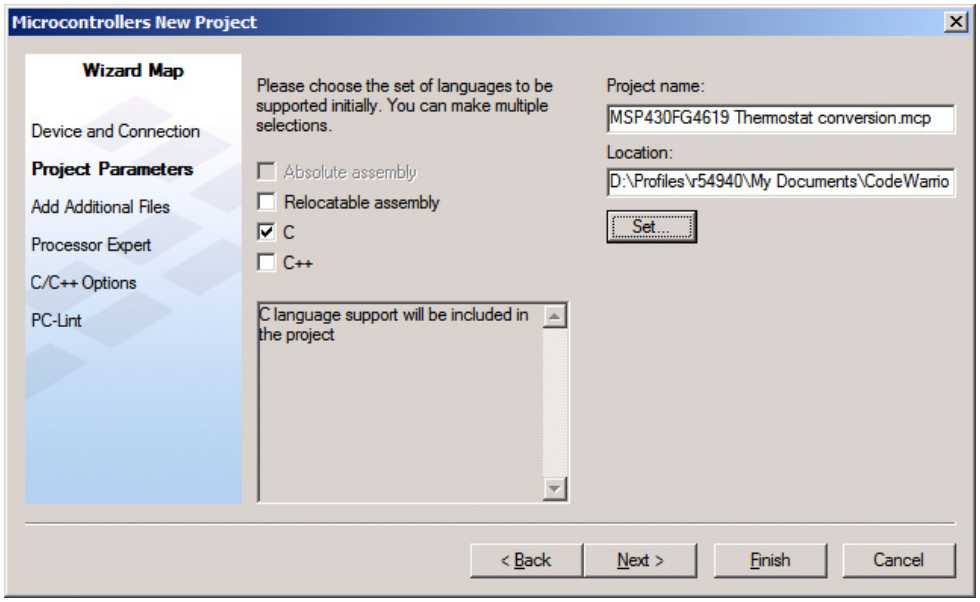

Figure 8. Language and Project Name

The project is built and the project window opened, which appears similar to the below image.

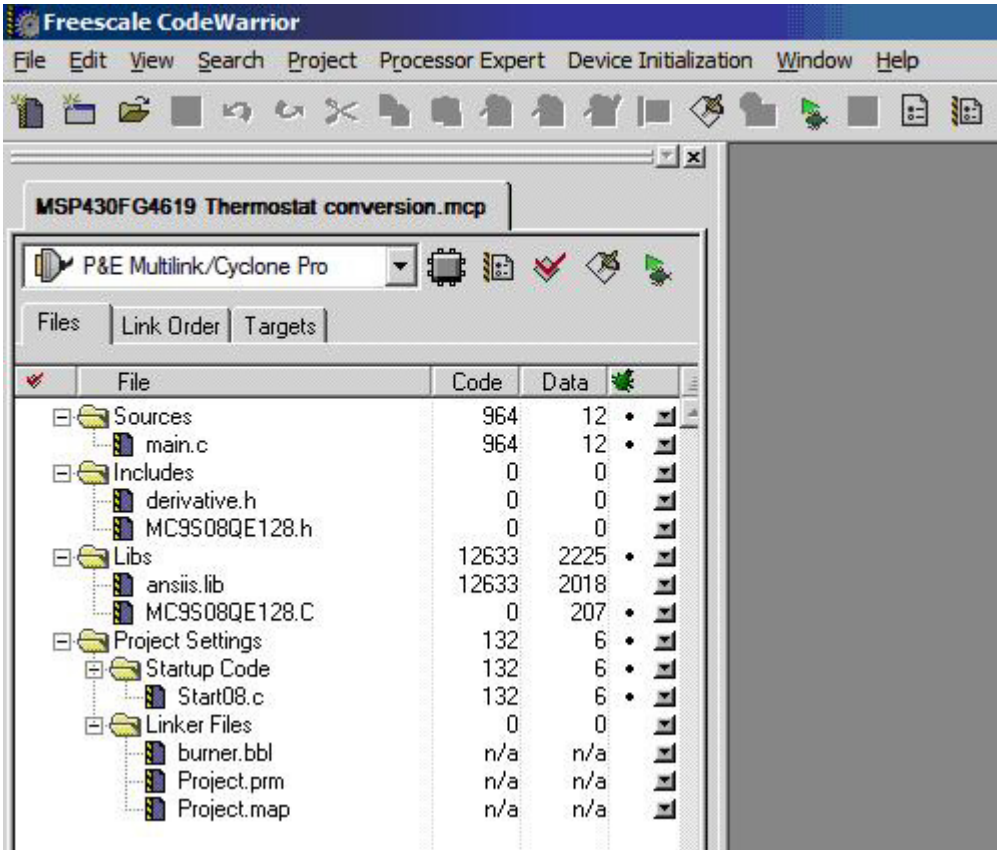

Figure 9. Codewarrior Project Window

If this is the first time that CodeWarrior has been used as a development environment, studying the QuickStart and AN2616, although it is written for an older CodeWarrior version, could be beneficial.

3.2.3 Move and Translate Code Following Flow Charts

AN3502 complements this application note because it describes the differences between the MSP430FG4619 and the QE128 Flexis devices on a modular and core level. In response, this section describes a method to aid conversion rather than a detailed description of all the changes. The full code for the MSP430FG4619 and the MC9S08QE128 is enclosed in Appendix A.

In the CodeWarrior environment, expressions that it does not recognise remain in black text, whereas others that are understood turn blue or green after the first compile (standard settings).

3.2.3.1 Header Files

CodeWarrior includes a MC9S08QE128.h file with definitions for all the registers, bits and masks needed for the MC9S08QE128, using unions to define all of the registers. The ADCCFG register is shown for demostration purposes in Example 1.

Example 1. ADCCFG definition from MC9S08QE128.h

```
/*** ADCCFG - Configuration Register; 0x00000016 ***/
typedef union {
  byte Byte;
  struct {
    byte ADICLK0     :1;                                       /* Input Clock Select Bit 0 */
    byte ADICLK1     :1;                                       /* Input Clock Select Bit 1 */
    byte MODE0       :1;                                     /* Conversion Mode Selection Bit 0 */
    byte MODE1       :1;                                     /* Conversion Mode Selection Bit 1 */
    byte ADLSMP      :1;                                      /* Long Sample Time Configuration */
    byte ADIV0       :1;                                           /* Clock Divide Select Bit 0 */
    byte ADIV1       :1;                                           /* Clock Divide Select Bit 1 */
    byte ADLPC       :1;                                            /* Low Power Configuration */
  } Bits;
  struct {
    byte grpADICLK :2;
    byte grpMODE :2;
    byte         :1;
    byte grpADIV :2;
    byte         :1;
  } MergedBits;
} ADCCFGSTR;
extern volatile ADCCFGSTR _ADCCFG @0x00000016;
#define ADCCFG                          _ADCCFG.Byte
#define ADCCFG_ADICLK0                  _ADCCFG.Bits.ADICLK0
#define ADCCFG_ADICLK1                  _ADCCFG.Bits.ADICLK1
#define ADCCFG_MODE0                    _ADCCFG.Bits.MODE0
#define ADCCFG_MODE1                    _ADCCFG.Bits.MODE1
#define ADCCFG_ADLSMP                   _ADCCFG.Bits.ADLSMP
#define ADCCFG_ADIV0                    _ADCCFG.Bits.ADIV0
#define ADCCFG_ADIV1                    _ADCCFG.Bits.ADIV1
#define ADCCFG_ADLPC                    _ADCCFG.Bits.ADLPC
#define ADCCFG_ADICLK                   _ADCCFG.MergedBits.grpADICLK
#define ADCCFG_MODE                     _ADCCFG.MergedBits.grpMODE
#define ADCCFG_ADIV                     _ADCCFG.MergedBits.grpADIV

#define ADCCFG_ADICLK0_MASK             1
#define ADCCFG_ADICLK1_MASK             2
#define ADCCFG_MODE0_MASK               4
#define ADCCFG_MODE1_MASK               8
#define ADCCFG_ADLSMP_MASK              16
#define ADCCFG_ADIV0_MASK               32
#define ADCCFG_ADIV1_MASK               64
#define ADCCFG_ADLPC_MASK               128
#define ADCCFG_ADICLK_MASK              3
#define ADCCFG_ADICLK_BITNUM            0
#define ADCCFG_MODE_MASK                12
#define ADCCFG_MODE_BITNUM              2
#define ADCCFG_ADIV_MASK                96
#define ADCCFG_ADIV_BITNUM              5
```

This register declaration makes the implementation very easy by using instructions like:

```
ADCCFG = 0x05;  // 8-bit Access to register
```

```
ADCCFG_ADLSMP = 1; // Bit access to register
ADCCFG_MODE = 2; // Group access to register (2-bit in example)
ADCCFG = ADCCFG_ADLSMP_MASK | ADCCFG_ADICLK0_MASK; // Use of masks for clearer code
```

The IAR Workbench development environment for the MSP430FG4619 uses the MSP420xG46x.h file. The ADC12CTL0 register is shown for demostration purposes in Example 2.

Example 2. ADC12CTL0 definition in MSP430xG46X.h

```
#define ADC12CTL0_          (0x01A0)  /* ADC12 Control 0 */
DEFW(   ADC12CTL0         , ADC12CTL0_)

/* ADC12CTL0 */
#define ADC12SC             (0x001)   /* ADC12 Start Conversion */
#define ENC                 (0x002)   /* ADC12 Enable Conversion */
#define ADC12TOVIE          (0x004)   /* ADC12 Timer Overflow interrupt enable */
#define ADC12OVIE           (0x008)   /* ADC12 Overflow interrupt enable */
#define ADC12ON             (0x010)   /* ADC12 On/enable */
#define REFON               (0x020)   /* ADC12 Reference on */
#define REF2_5V             (0x040)   /* ADC12 Ref 0:1.5V / 1:2.5V */
#define MSC                 (0x080)   /* ADC12 Multiple SampleConversion */
#define SHT00               (0x0100)  /* ADC12 Sample Hold 0 Select 0 */
#define SHT01               (0x0200)  /* ADC12 Sample Hold 0 Select 1 */
#define SHT02               (0x0400)  /* ADC12 Sample Hold 0 Select 2 */
#define SHT03               (0x0800)  /* ADC12 Sample Hold 0 Select 3 */
#define SHT10               (0x1000)  /* ADC12 Sample Hold 0 Select 0 */
#define SHT11               (0x2000)  /* ADC12 Sample Hold 1 Select 1 */
#define SHT12               (0x4000)  /* ADC12 Sample Hold 2 Select 2 */
#define SHT13               (0x8000)  /* ADC12 Sample Hold 3 Select 3 */
#define MSH                 (0x080)

#define SHT0_0               (0*0x100u)
#define SHT0_1               (1*0x100u)
#define SHT0_2               (2*0x100u)
#define SHT0_3               (3*0x100u)
#define SHT0_4               (4*0x100u)
#define SHT0_5               (5*0x100u)
#define SHT0_6               (6*0x100u)
#define SHT0_7               (7*0x100u)
#define SHT0_8               (8*0x100u)
#define SHT0_9               (9*0x100u)
#define SHT0_10             (10*0x100u)
#define SHT0_11             (11*0x100u)
#define SHT0_12             (12*0x100u)
#define SHT0_13             (13*0x100u)
#define SHT0_14             (14*0x100u)
#define SHT0_15             (15*0x100u)

#define SHT1_0               (0*0x1000u)
#define SHT1_1               (1*0x1000u)
#define SHT1_2               (2*0x1000u)
#define SHT1_3               (3*0x1000u)
#define SHT1_4               (4*0x1000u)
#define SHT1_5               (5*0x1000u)
#define SHT1_6               (6*0x1000u)
#define SHT1_7               (7*0x1000u)
#define SHT1_8               (8*0x1000u)
```

```
#define SHT1_9                (9*0x1000u)
#define SHT1_10              (10*0x1000u)
#define SHT1_11              (11*0x1000u)
#define SHT1_12              (12*0x1000u)
#define SHT1_13              (13*0x1000u)
#define SHT1_14              (14*0x1000u)
#define SHT1_15              (15*0x1000u)
```

This register declaration makes the implementation possible by using instructions like:

```
ADC12CTL0 = 0x0000;  // 16-bit Access to register
ADC12CTL0 = ADC12CTL0|SHT01 // Bit access to register
ADC12CTL0 = ADC12CTL0|SHT0_1// "Group" access to register (2-bit in example)
ADC12CTL0 = ADC12CTL0|ENC|ADC12SC; // Use of masks for clearer code
```

but not quite as easy to read.

3.2.3.2 Definitions and Declarations

The first sections that should be transferred across software platforms, because they are not device dependant, are the definitions of the constants, global variables, and functions. Then manage the placement in memory of the set_point pointer.

3.2.3.3 Main Body

Starting at main, copy over the first chunk of device specific code, which is the the disable interrupts line and the initialize function in this example.

The _BIC_SR(GIE) can be replaced with the SIE instruction in the S08 core. However, for compatibility with the MCF51QE128 device, the DisableInterrupts macro should be used.

3.2.3.3.1 Initialize

The initialize routine touches on five of the main modules used by the program. Because the register names are device specific, each line must be translated using the information in the reference manuals to find the equivalent setting.

The internal clock source (ICS) module should also be configured (Example 3) to run at the highest possible frequency. You can choose an internal or external oscillator. However, because LPR mode must use the external oscillator, it is better to enable it at the beginning rather than switching between the two and loosing CPU cycles while the status flags update.

Example 3. ICS configuration on QE128

```
void configureICS(void)
{
  /* FEE @ 50.33MHz CPU = Bus*2 */
  ICSC2 = 0x07;              /* Low Range, Low Power, BDIV_1, Ext ref selected */
  ICSC1 = 0x00;              /* FLL Enabled, int osc disabled */
  ICSSC_DRST_DRS = 0x2;
  ICSSC_DMX32 = 0;           /* FLL factor of 1536 */
  while (ICSSC_IREFST == 1); /* Wait for ext clock as ref indicator */
}
```

The Basic Timer function on the MSP430FG4619 (Example 4) can be replaced with the RTC module on the QE128. The module must function to generate a wake-up interrupt every second. The RTC module needs only one register to be configured. The status and control register (RTCSC) as the counter is read only and the modulus is used in its default condition as seen in Example 5.

RTCSC controls the clock source, interrupts and the prescaler. The lowest power clock source option is the LPO, but accuracy is an issue in temperature varing clock applications. The OSCOUT (32.768kHz) signal is the better option and allows operation in Stop3 mode. To use this source, the external clock must be enabled in the ICS. The prescaler can be set at 215 (RTCPS = 6) to generate a 1 second period, meaning the modulus can be left at zero. Later in the MSP430FG4619 code, the counter is written to close the time between Run and Display. The RTC counter cannot be written to so this feature cannot be reproduced.

Example 4. Basic Timer Init on MSP430FG4619

```
// Configure basic timer to interrupt ~1 second */
  BTCTL = BT_ADLY_1000;    /* ~1 second */
IE2 = BTIE;              /* Enable Basic Timer interrupts */
```

Example 5. RTC Init on QE128

```
/* Configure RTC to interrupt ~1 second */
RTCSC = 0x36;     /* Enable interrupts and use OSCOUT to generate a 1 second period. */
```

The next section of the initialize routine is the SPI module. The QE128 does not need to specifically enable the SPI module on the pin because the function is governed by the pin sharing priority rules shown in the Pins and Connections chapter of the MC9S08QE128 Reference Manual. The registers in the MSP430 SPI and the QE128 SPI are not compatible. Therefore, well commented code is the key because the comments enable the creation of the same function on the QE128 device. Example 6 and Example 7 show the two different SPI initialisations.

Example 6. USCI_A0 SPI Init on MSP430FG4619

```
/* Initialize SPI (using USCI_A0)  */
  UCA0CTL1 = 0x01;                      /* Hold in reset state */
  UCA0CTL0 = 0x6B;                      /* Master - MSB first - 4pin idle high
                                     for slave transmission */
  UCA0CTL1 = 0x41;                      /* Select ACLK as USCI clock source and
                                     hold in reset state */
  UCA0BR0 = 0x01;
  UCA0BR1 = 0x00;                       /* Set SPI Bit Rate = Module Clock */
  IFG2 = 0x00;                          /* Clear all flags */
  P7SEL = 0x0F;                         /* Enable SPI pins */
  P7DIR = 0xFF;                         /* Set port to outputs */
  IE2 = IE2|UCA0TXIE;                   /* Enable Tx interrupt on USCIA0/SPI */
```

Example 7. SPI Init on QE128

```
  /* Initialize SPI (using SPI2) */
  SPI2C1 = 0x1E;            /* Master, Tx ISR enabled, MSB First, Idle Low,
                          First Edge beginning first cycle, SPI disabled */
  SPI2C2 = 0x10;            /* Normal state, Automatic SS */
  SPI2BR = 0x00;            /* Max Bus Clock */
  SPI2S;                   /* Read Status register to clear flag (dummy) */
  SPI2C1_SPE = 1;           /* Enable SPI 2 */
```

After the SPI initialization is the initialization of the pins. Follow the recommended configuration of unused I/O. You can choose to include the KBI for the switches here or in a separate section. Next, tackle the ADC configuration. Initialize is now converted.

3.2.3.3.2 Button Presses

Following the flow chart of the main routine, the next chunk of code to be converted should be the state button loop.

Back in the main loop, the code up to the point where the stateButtonPressed function is called should be copied to the new project.

By changing the if statement to be QE128 compatible (if [P1IFG &= 0x02] changed to if [KBI2SC_KBF && PTDD_PTD45]), the setButtonPressed routine can then be copied into the project. Because none of the immediate code is MCU dependant, no other software changes are required.

The handleMode Button routine calls one function, switchDelay(), which is also called in other sections of the code. You can choose to convert it now or convert it later. Because the next function is similar to the lines that were recently converted, it is advisable to continue with the setButtonPressed routine.

By changing the setButtonPressed if statement to be QE128 compatible (if [P1IFG &= 0x04] changed to if [KBI2SC_KBF && PTDD_PTDD5]), the setButtonPressed routine can then be copied into the project. The setButtonPressed routine calls two functions; switchDelay() and incMinute().

IncMinute does not require any code conversion because it uses no MCU module function and can be copied straight across.

The switchDelay uses a timer to protect against switch bounce. This places the device in to sleep mode (LPM3) for 200ms (using ACLK as a timebase) and wakes it by the TimerA compare interrupt on the MSP430FG4619 (Example 8). The RTC cannot be reused because it is keeping time. Because the timers on the QE128 cannot wake up from stop modes, a wait mode must be used instead (Example 9). Because low-power wait can become full-run mode and the RTC is using the OSCOUT clock source (ICSOUT independent), this wait mode can be used. The decision to use this mode means that debugging is limited because the debug modules must be off in this mode. By setting up the timer, the ICS and the system registers can convert the code as showin in Example 8.

Example 8. MSP430FG4619 SwitchDelay Function

```
void switchDelay(void)
{
  TACCTL0 = CCIE;                          /* Enable CCR0 interrupt */
  TACCR0 = 6550 - 1;                       /* Value reqiured for delay of ~200mS */
  TACTL = TASSEL_1+TACLR +MC_1; /* ACLK=32kHz; TAR=0; up state and start Timer */
  _BIS_SR(LPM3_bits + GIE);                /* Go to sleep */
  /* Zzzzzzzzzzzzz */
  TACTL = 0;                               /* Disable timer */
  TACCR0 = 0;
  P1IFG = 0x00;                            /* Clear Flags */
  BtnHandlerFlag = 0;                      /* Allow RTC Interrupts */
}
```

Example 9. QE128 SwitchDelay Function

```
void switchDelay(void)
```

```
{
  /*  Delay to prevent re-run due to switch bounce */
  SPMSC2_LPWUI = 1;/* Enable wake from wait to full run state */
  TPM2SC = 0x48;              /* Overflow int enabled and BusClk as source */
  TPM2MOD = 0x1900;/* Mod of 6400 @ 32kHz is 200ms */
  enterLPR();
  _Wait;                   /* Enter LPW */
  /* Zzzzzzzzzzzzz */
  TPM2SC = 0x00;              /* Disable timer */
  TPM2MOD = 0;    /* Clear Modulus */
  configureICS();             /* Get CPU back up to maximum speed */
  KBI2SC_KBACK = 1;           /* Clear KBI Flag */
  BtnHandlerFlag = 0;         /* Allow RTC_ISR to exit Stop3  */
}

void enterLPR(void)
{
   ICSC1_IREFS = 0; /* select Ext ref clock */
   ICSC1_CLKS = 2;
   ICSC2_LP = 0;
   ICSC2_BDIV = 0;       /* Run at 32kHz CPU - BDIV / 1 */
   ICSC2_LP = 1;               /* Enter FBELP */
   while (ICSSC_IREFST && (ICSSC_CLKST != 0x10));
         /* Wait for status bits to update then enter LPR */
   SPMSC1 ^= 0x0C;/* Clear LVDE ane LVDSE */
   SPMSC2_LPR = 1;/* Enter LPR state */
}
```

The details on how the device mode transitions differ between the MC9S08QE128 and the MSP430FG4619, which have been included in Example 8 and Example 9, are explained in Section 3.2.3.6.1, “Mode Transitions”.

Figure 11-7 in the MC9S08QE128 Reference Manual shows how to tranistion around the ICS modes legally and, used with Figure 3-1 in the same document, is referenced in creation of the QE128 code.

That completes the state and the set button loops of Figure 4.

3.2.3.3.3 Time Controlled Rtherm Reading, Display Update, and KBI Check

Now that the button press code is converted, the SecCycleFlag can be managed. This code section should be run every iteration in fault-free operation because it is set in the RTC ISR, the only LPM3/Stop3 wake up source. This section manages the state time and does not require any re-work.

The remainder of the main loop code can be copied over and CodeWarrior highlights that the ADC functionality, displayState() and the KBI must be translated.

The ADC has already been initialized so it only needs to be started and the Conversion Complete flag polled.

Displaystate() is a function that doesn't require any modification because it only calls further functions, displayInt, depending on the current state. To aid in debugging, two LEDs can be added (Example 10) to add meaning to the state dependant digits being transmitted.

Example 10. Debugging assistance; Mode display

```
/* Display to user the state being transmitted without affecting Heat LED */
MSP430FG4619      -> P1OUT = ((P1OUT&0x01)|(state<<6));
MC9S08/MCF51QE128 -> PTED = (((char)PTED_PTED0)|(state<<6));
```

The displayInt function is the same, but it calls the transmit function that engages the SPI. By managing the error flags and changing the registers to the QE128 function, the function is translated (Example 11 and Example 12).

Example 11. Transmit Function on MSP430FG4619

```
void transmitSPI (signed char *textPointer)
{
  while(*textPointer != 0)               /* While not end of string */
    {
    UCA0CTL1 = 0x40;                     /* Release USCIA0 from reset state */
    UCA0TXBUF = *textPointer;   /* Write the character to the USCIA0 interface */
    while ((UCA0STAT&UCBUSY) == 1);     /* Wait until transmission is complete */
    UCA0RXBUF;                           /* Read Rx Buffer to prevent SPI error */

    if (UCA0STAT != 0)                   /* If an error flag is set */
      {
      UCA0TXBUF = *textPointer;          /* Re-transmit the message */
      while ((UCA0STAT&UCBUSY) == 1);    /* Wait until re-trans is complete */
      }

    UCA0CTL1 = 0x41;                     /* Place USCIA0 in reset state */
    textPointer += 1;/* Increment to point at the next character in the string */
    }
}
```

Example 12. Transmit Function on QE128

```
void transmitSPI (signed char *textPointer)
{
    while(*textPointer != 0)         /* While not end of string */
     {
     SPI2C1_SPE = 1;                 /* Enable SPI 2 */
     while (SPI2S_SPTEF == 0);       /* Wait until Transmit buffer is empty */

     /* write the character to the SPI2 interface */
     SPI2D = *textPointer;
     while (SPI2S_SPTEF == 0);         /* Wait until transmission is complete */

     if (SPI2S_MODF != 0)              /* If an error flag is set */
        {
        SPI2S_MODF;                    /* Clear SPI error flag */
        SPI2D = *textPointer;          /* Re-transmit the message */
        while (SPI2S_SPTEF == 0);     /* Wait until re-transmission is complete */
        }
     /* increment to point at the next character in the string */
     textPointer += 1;
     SPI2C1_SPE = 0;              /* Disable SPI 2 */
     }
}
```

The KBI check at the end of the main loop can be converted thanks to Example 13 and Example 14.

Example 13. Confirm KBI Flags All Clear on MSP430FG4619

```
if (P1IFG == 0)                          /* If all KBI flags handled */
    {
    _BIS_SR(LPM3_bits + GIE);            /* Re-enable interrupts and go to sleep */
    } /* Else, MCU will run main loop again until all flags cleared */
```

Example 14. Confirm KBI Flags All Clear on QE128

```
if (!KBI2SC_KBF)                         /* If all KBI flags handled */
   {
   EnableInterrupts;                     /* Re-enable pin interrupts */
   enterStop3();           /* Go to sleep */
   } /* Else, MCU will run main loop again until all flags cleared */

void enterStop3(void)
   {
   SPMSC1 ^= 0x0C;                   /* Clear LVDE ane LVDSE */
   _Stop;
   }
```

3.2.3.4 Close Loose Ends

After all the code is converted, the function declarations can be checked to ensure that all required functions have been copied in and are linked correctly. The following three functions are new for the QE128 and must be added in the declaration section.

```
void configureICS(void);        /* Routine to configure ICS for full run state */
void enterLPR(void);  /* Handles ICS and SPMSC registers */
void enterStop3(void);
```

3.2.3.5 ISRs

The standard CodeWarrior project includes only the reset ISR, `VECTOR 0  _Startup /* Reset vector: this is the default entry point for an application. */`, in the .prm file, so now that the main program is complete the other interrupt vectors must be inserted.

The application requires the inclusion and conversion of three other interrupt routine/functions. The VectorNumber_Vname interrupt declarations are defined in the CodeWarrior header files and can be placed in the main c file as shown in Example 15.

This method allows for easy coversion to MCF51QE128 at a later time.

Example 15. Interrupt Service Routines on QE128

```
interrupt VectorNumber_Vtpm2ovf void _TPM2_OVF_ISR(void)
{
   TPM2SC;                 /* Dummy Read */
   TPM2SC_TOF = 0;/* Clear Flag */
         }

interrupt VectorNumber_Vspi2 void _SPI2_ISR(void)
{
   if (SPI2S_SPRF)
      {
      SPI2S;      /* Dummy Read */
      SPI2D;      /* Dummy Read to clear flag */
```

```
      }
   if (SPI2S_SPTEF);/* If Tx Buffer Empty Flag set ignore and cont.. */
   if (SPI2S_MODF)
      {
      SPI2S;      /* Dummy Read */
      displayState();/* Retramsit message */
      }
}

interrupt VectorNumber_Vrtc void _RTC_ISR(void)
{
   RTCSC_RTIF = 1;           /* Clear Flag */
   /* Handle the clock tick in the ISR to ensure it happens immediately. */
   second++;
   if (second >= 0x60)
      {
      second = 0;
      incMinute();
      }
   SecCycleFlag = 1;         /* Set flag to handle in main() next time it runs */
   if (!(BtnHandlerFlag))   /* Don't int debounce timer in the button handlers */
   KBI2SC_KBIE = 0;          /* Prevent KBI from disrupting */
}
```

3.2.3.6 System Control

The QE128 project can now be compiled. The reported dummy read errors can be turned off by placing `#pragma MESSAGE DISABLE C4002 /* Disable Result Not Used warning */` at the beginning of the main file, before all the defines. The condition always true can be disabled by `#pragma MESSAGE DISABLE C4000 /* Disable Condition Always True warning */`.

3.2.3.6.1 Mode Transitions

AN3502, Differences Between the TI MSP430 and MC9S08QE128 and MCF51QE128 Flexis Microcontrollers, discusses the CPU modes of the two devices and how they compare functionally, but does not touch on the practical implications.

The six modes of the MSP430 CPU are controlled via four bits; SCG0, SCG1, OSCOFF, and CPUOFF.

Table 2. Modes of the MSP430

SCG1	SCG0	OSC0FF	CPU0FF	MODE	CPU and Clock Status
0	0	0	0	Active	CPU is active, all enabled clocks are active
0	0	0	1	LPM0	CPU and MCLK are disabled (41x/42x peripheral MCLK remains on) SMCLK and ACLK are active
0	1	0	1	LPM1	CPU, MCLK, and DCO osc are disabled (41x/42x peripheral MCLK remains on) DC generator is disabled if the DCO is not used for MCLK or SMCLK in active mode SMCLK and ACLK are active
1	0	0	1	LPM2	CPU, MCLK, SMCLK, and DCO osc. are disabled DC generator remains disabled ACLK is active
1	1	0	1	LPM3	CPU, MCLK, SMCLK, and DCO osc. are disabled DC generator remains disabled ACLK is active
1	1	1	1	LPM4	CPU and all clocks disabled

Because all low-power modes have the CPU-disabled transitions between them are not possible. Wake up is via interrupts so bit-set (BIS) and bit-clear (BIC) assembly instructions can be used to control these five bits. The intrinsics.h defines functions for low-power mode entry so that higher level commands can be used increasing code readability as shown in Example 16.

Example 16. CPU Mode Function Declarations in Intrinsics.h

```
/* Functions for controlling the processor operation modes */

#define __low_power_mode_0() (__bis_SR_register(  __SR_GIE      \
                                                | __SR_CPU_OFF))

#define __low_power_mode_1() (__bis_SR_register(  __SR_GIE      \
                                                | __SR_CPU_OFF  \
                                                | __SR_SCG0))

#define __low_power_mode_2() (__bis_SR_register(  __SR_GIE      \
                                                | __SR_CPU_OFF  \
                                                | __SR_SCG1))

#define __low_power_mode_3()                    \
  (__bis_SR_register(  __SR_GIE                 \
                     | __SR_CPU_OFF             \
                     | __SR_SCG0                \
                     | __SR_SCG1))

#define __low_power_mode_4()                    \
  (__bis_SR_register(  __SR_GIE                 \
                     | __SR_CPU_OFF             \
                     | __SR_SCG0                \
                     | __SR_SCG1                \
                     | __SR_OSC_OFF))
```

```
#define __low_power_mode_off_on_exit()          \
  (__bic_SR_register_on_exit(  __SR_CPU_OFF     \
                             | __SR_SCG0        \
                             | __SR_SCG1        \
                             | __SR_OSC_OFF))
```

The MC9S08QE128 controls low-power mode entry through two CPU instructions stop and wait. This means that mode entry is faster, but time must be taken beforehand to set up LVDE, LVDSE, LPR, and PPDC in the system power management and control registers (SPMSC1 and SPMSC2).

Table 3. Operating Mode Conditions

Mode of Operations	BDCSCR BDM	SPMSC1 PMC		SPMSC2 PMC		CPU and Peripheral CLKs	Effects on Sub-System	
	ENDBDM	LVDE	LVDSE	LPR	PPDC		BDM Clock	Voltage Regulator
Run mode	0	X	X	0	X	On. ICS in any mode	Off	On
		1	1	1				
	1	X	X	X			On	
LPRUN	0	0	X	1	0	Low frequency required. ICS in FBELP mode only	Off	Standby
WAIT mode Assumes WAIT instruction executed	0	X	X	0	X	CPU clock is off; Peripheral clocks on. ICS state same as RUN mode.	Off	On
		1	1	1				
	1	X	X	X			On	
LPWAIT mode Assumes WAIT instruction executed	0	0	X	1	0	CPU clock is off; Peripheral clocks at low speed. ICS in FBELP mode.	Off	Standby
		1	0					
STOP3[1] Assumes STOPE bit is set and STOP instruction executed.	0	0	X	X	0	ICS in STOP, LPO, OSCOUT, ICSERCLK, and ICSIRCLK optionally on	Off	Standby
	0	1	0	X	0		Off	
	0	1	1	X	X		Off	On Stop currents are increased
	1	X	X	X	X		On	
STOP2[1] Assumes STOPE bit is set and STOP instruction executed.	0	0	X	0	1	LPO and OSCOUT optionally on	Off	Partial Powerdown
		1	0					

[1] STOP3 is used in place of STOP2 if the BDM or LVD is enabled.

You can create functions that set and clear the correct bits for each mode, which would be beneficial if modes are transitions frequently to save code memory.

```
void enterStop3(void)
{
   SPMSC1 ^= 0x0C;       /* Clear LVDE ane LVDSE */
   _Stop;
}
```

ENBDM is automatically set or clear with the detection of BDM communications (or lack of) on the BDM pin. Using the _Stop and _Wait macros rather than the assembly instruction allows easy transition to CFV1 because Codewarrior manages the core differences.

The CPU of the CFV1 has only one low-power mode, stop. Wait is a synthetic mode controlled by logic. The SOPT register has two bits, instead of one, that control the effect of the stop instruction. The STOPE write-once bit enables stop mode. The WAITE write-any time bit enables wait mode. More details can be found in AN3460, Low-Power Design Enabled by MC9S08QE128 and MCF51QE128 Flexis Microcontrollers, and the MCF51QE128 Reference Manual.

3.2.3.6.2 System Registers

After the file compiles the QE128, specific system registers must be set before the project runs on silicon. A good place to do this is at the start of the initialize function.

SOPT1, SOPT2, SPMSC1, SPMSC2, and SPMSC3 control the MCU system and are described in Chapter 5 of the Reference Manual.

SOPT1 is a write-once register and should be written soon after the reset vector to avoid undesired activity. SOPT1 controls the COP, Stop modes (and Wait mode on the MCF51QE128), and the reset and the background pins. The code sets the register as 0x23: stop mode enabled, background, and reset pin enabled.

- SOPT2 controls the COP clock and some of the module settings. The default setting is used in code.
- SPMSC1 controls the low voltage options and the bandgap buffer. The default setting is used in code.
- SPMSC2 controls the low-power mode settings and is left as default setting updated as required in the code body.
- SPMSC3 is the second register to control and report the low voltage status. The default setting is used in code.

NOTE

It is advisable to match the stack size to the MSP430FG4619 stack in this project.prm file because stack overflow errors are hard to recognise when you are unfamiliar with a device and debugging environment.

3.2.4 Add New Features, When Available

After the application has been converted and is confirmed to be working as expected, new features can be added to reduce power or increase performance by taking advantage of the new target MCUs features.

3.2.4.1 Clock Gating

The QE128 devices have a clock gating feature that disables the clocking signal to various peripheral modules independently from the CPU modes. By gating off the clock to unused modules, precious micro amps (µA) can be saved in the MCU's run and wait modes. This feature is especially important because of the numerous communications modules and timers incorporated in the QE128. Applications are unlikely to use all of them. Benefits can be found only by gating the modules on in the section of the application that needs them and by gating them off when the job is done.

Be careful when using this feature because all the clocks are gated on after a reset. To keep power consumption down, the clocks should be gated off as soon as possible. Writes to registers associated with a gated off module have no effect (e.g. writing to the Modulo of the RTC module if the SCGC2_RTC bit is 1 does not cause the register value to change). To avoid errors, disable the module before gating it off and re-initialize the registers when the module is gated back on.

- The ADC is used only every 30 seconds.
- The SPI is used every second.
- TPM2 is only used during the switch debounce delay.
- The Debug module clock should be gated on during project development but can be gated off for production and testing.
- The KBI module could be used at any point in the application so should never be gated off.
- The RTC is always running so should never be gated off.

The code in Example 17 is the same as Example 8 but with clock gating added.

Example 17. Switchdelay Function with Clock Gating Added

```
void switchDelay(void)
{
  SCGC1_TPM2 = 1;             /* Gate on TPM2 */
  /*  Delay to prevent re-run due to switch bounce */
  SPMSC2_LPWUI = 1;/* Enable wake from wait to full run state */
  TPM2SC = 0x48;              /* Overflow int enabled and Bus as module source */
  TPM2MOD = 0x1900;/* Mod of 6400 @ 32kHz is 200ms */
  enterLPR();
  _Wait;                  /* Enter LPW */
   /* Zzzzzzzzzzzzz */
  TPM2SC = 0x00;              /* Disable timer */
  TPM2MOD = 0;    /* Clear Modulus */
  configureICS();             /* Get CPU back up to maximum speed */
  KBI2SC_KBACK = 1;           /* Clear KBI Flag */
  BtnHandlerFlag = 0;         /* Allow RTC_ISR to exit Stop3  */
  SCGC1_TPM2 = 0;             /* Gate off TPM2 */
}
```

Because gating on and off the module recommends re-initialing the associated registers, the ADC and SPI initialization can be extracted from the initial MCU function into their own functions and called when required, remembering to add them to the function declarataion section.

The MSP430FG4619 has the ability to gate off MCLK and SMCLK at the source, i.e in the FLL+ module, but not on a per module basis.

3.2.4.2 Supply Voltage Warning

The example application used in this application note does not make use of the LVD (QE128) and SVS (MSP430FG4619) systems because the thermostat would most likely be AC powered. However, in battery applications, the QE128's low voltage warning flag (LVW) could be used to send a warning indicator to and place the application into a safe state before the critical voltage was reached.

3.2.4.3 Port Manipulation

The QE128 has port manipulation functions on Ports C and E. Therefor, the example application used in this application note uses Port E bits 0, 6, and 7 as outputs to LEDs. The port set, clear, and toggle registers can be used instead of direct data register manipulation to speed up the execution time of the main loop (Example 17). This reduces power consumption as more time is spent in low-power mode.

Example 18. Port Manipulation on Heat LED

```
if (currentTemp<setTemp)
   PTESET = 0x01;               /* Turn on Heat LED */
else
   PTECLR = 0x01;                        /* Turn off Heat LED */
```

3.2.4.4 Modes of Operation

At this stage the application code is converted and highly optimized to take advantage of the QE128's features. One last thing that can be checked is that the best ICS and CPU modes are used in the right places to improve the power consumption.

The QE128 code currently uses three modes:

- Run mode in FEE @ 50.33MHz CPU speed, 25.165MHz bus.
  - Predominant mode for main loop.
  - Fastest speed available
  - Typical Idd around 15mA
- Low Power Wait mode @ 32kHz CPU, 16kHz bus, FBELP
  - Used for 200ms each button press.
  - Typical Idd <10uA
- Stop3 with LVDE and LVDSE clear
  - Lowest power available when BDM is off.
  - Used for dormant state between 1 seconds wake ups from RTC.
  - Typical Idd around 1.5uA (with RTC/ERCLK adder)

The main loop should be run as fast as possible to maximise the time in Stop3 mode. Using the external clock source reduced the MCU's power consumption but not necessarily the systems power consumption. Experiments can be carried out to find the best oscillator source for your application.

Low-Power wait mode can run down to 4 kHz CPU/2 kHz Bus. However, reducing the speed down to this speed may not have much power saving advantage to the system because low power wait (LPW) mode is

only used when a button is pressed. Also, the CPU takes longer to execute the code to bring it back to 50.33 MHz. In applications that use LPW more extensively, this modification make more sense.

Stop3 mode (when used with the BDM off) is a low-power mode. In this case, Stop2 cannot be used due to the periodic wake up nature of the system and the ERCLK must be used for time keeping accuracy. Other applications may be able to benefit from using different RTC clocks or module wake-ups to improve power consumption.

3.2.5 Conversion to MCF51QE128

This code has been written to be S08 and CFV1 compatible. However, to make the code in Appendix A.2 run on MCF51QE128, the interrupt wake-up must be enabled at the start of main() and the set pointer moved to the new RAM start location at 0x0080_0000 in the defines.

```
INTC_WCR = 0x80;              /* Enable Interrupt wake up signals */
char *set_point = (char *)0x00800000;   /* set temp pointer located in RAM @ 0x0080_0000 */
```

AN3465, Migrating Within The Controller Continuum, is available to help write code that runs on the MC9S08QE128 or the MCF51QE128.

4 Power Comparison

The measurements in Table 4 were taken on the previously described boards with unused pins terminated as recommended by the manufacturer using the same power supply and the same ammeter. The rest of the code was as-is so any additional current due to operating modules is included.

Table 4. Power Consumption of System in each Application State

Device	Mode	Comment	Idd	Time in Mode
MC9S08QE128	Run	Executed every second	4.7 mA	123 us
MCF51QE128	Run	Executed every second	10 mA	48 us
MSP430FG4619	Active	Executed every second	3.2 mA	475 us
MC9S08QE128	LPW	Executed every button press	4.2 uA	200 ms
MCF51QE128	LPW	Executed every button press	3.5 uA	200 ms

Table 4. Power Consumption of System in each Application State (continued)

Device	Mode	Comment	Idd	Time in Mode
MSP430FG4619	LPM3	Executed every button press	3.2 uA	200 ms
MC9S08QE128	Stop3	Executed every second	1.1 uA	1000 ms
MCF51QE128	Stop3	Executed every second	1.2 uA	1000 ms
MSP430FG4619	LPM3	Executed every second	2.9 uA	995 ms

MC9S08QE128 Mask Set 1M11J, MCF51QE128 Mask Set 0M12J, and MSP430FG4619 Rev E were used.

The run/active mode IDD was measured by placing a while (1) in the main loop. The low-power mode Stop3/LPM3 IDDs are measured by disabling the RTC interrupt function whereas the LPW/LPM3 Idds are measured by removing the stop instruction and disabling the timer and RTC interrupts. All measurements are done with the BDm or JTAG tools disabled.

These IDDs are not guaranteed and are only indicative as one device was used for each measurement. The measurement was taken on the VDD supply from the MSP-FET430UIF tool for the MSP430FG4619 and through the MCU_VDD jumper on the EVBQE128 board. There is some error induced because the boards are different, but all possible measures were taken to remove any possible current drains.

The time spent in the mode was measured by setting and clearing a pin before and after the mode entry and exit and measuring the period with an oscilloscope.

Figure 10 shows the energy used by each device in each application mode in realation to the MC9S08QE128. The energy is a function of IDD (μA), time (ms), and voltage (V).

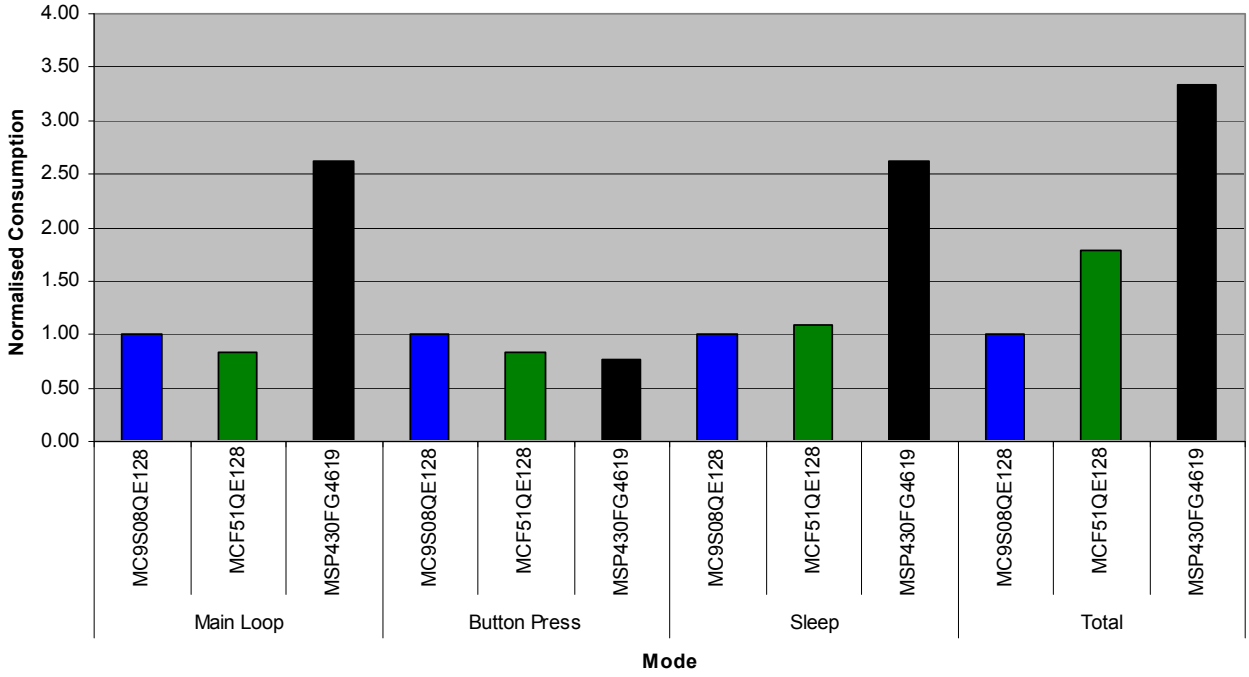

Figure 10. Energy Used in Each Device in Relation to MC9S08QE128

The QE128 devices have a significant advantage in the main loop due to their higher bus speed capability enabling a better µA per MHz ration. The MSP430FG4619 devices use of LPM3 instead of the QE128 LPW mode during the button press function gives this device the advantage here. However, the QE128 takes the advantage by being able to use Stop3 with only the RTC/ERCLK enabled in the sleep section.

5 Conclusion

The key to a successful conversion of an application from one device to another, in a different family or different core, is a solid understanding of the application software helped by good comments. After that level of knowledge is gained, time spent cleaning up the software aids conversion and allows for an easier conversion because your own coding style and tricks can be added. Drawing a flow chart can help you convert the code in a logical order and stop any routines or functions being left out. After the code is converted and is running the devices, the reference manual can be scoured to uncover new features that can be included to improve performance and power consumption.

Appendix A
Code Listing

A.1 MSP430FG4619 Code

```
/* Code by Inga Harris for MSP430FG4619  */
/* Freescale Semiconductor  */
/* September 2007  */
/* Built with IAR Embedded Workbench Version: 3.42A  */

#include <msp430xG46x.h>

/* Define states */
#define TEMP_state         0               /* Temperature state */
#define SET_state          1               /* Set target temp/time state */
#define TIME_state         2               /* Clock state */

/* Constants */
#define minimumT          10               /* 10 deg C */
#define maximumT          40               /* 40 deg C */
#define tempCheck_period  30               /* Measure temp every 30 seconds */

/* Clock values and initialisation */
unsigned char hour = 0x0C;     /* Hours: starts from poweron or reset @ midday */
unsigned char minute = 0x00;               /* Minutes (tens/ones) */
unsigned char second = 0x00;               /* Seconds (tens/ones) */

/* Temperature values and initialisation */
unsigned char currentTemp;                 /* Current Temperature */
unsigned char setTemp = 22;                /* Set Point Temperature to 22degC */
char *set_point = (char *)0x0200;          /* Set Temperature Pointer @ start of RAM */

unsigned char state = TEMP_state;          /* Indicates the user state and initialise */

/* Flags for communication between ISRs and main() and initialisation */
char BtnHandlerFlag = 0;                   /* Indicate btn press being handled */
char StateButtonFlag = 0;                  /* Run state rountine next loop itteration */
char SecCycleFlag = 0;                     /* Run 1sec cycle next loop itteration */
char SetButtonFlag = 0;                    /* Run set routine next loop itteration */

/* Temperature array */
const unsigned char Rtherm_array[31] = {10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
                                        20, 21, 22, 23, 24, 25, 26, 27, 28, 29,
                                        30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40};

/* Function declarations */
void displayState(void);                   /* Top level general display function */
void displayInt(unsigned int, unsigned char); /* Display numbers, includes ASCII conversion */
void incMinute(void);
void initializeMCU(void);
void stateButtonPressed(void);             /* Handles press of "state" switch */
void setButtonPressed(void);               /* Handles press of "set" switch */
void switchDelay(void);                   /* Short (200ms) delay to avoid switch debounce */
void transmitSPI (signed char *textPointer);/* Handles SPI transmission */
```

```
/*****************************************************************************/

main()
{
  unsigned int Tupdate_counter = tempCheck_period - 1;  /* Inialialise Tupdate_counter to remove
delay between application start and 1st display */
  unsigned int stateTime = 0;               /* Cycles spent in current state */
  unsigned char Rtherm_elem;                /* Temperature array element number */

  _BIC_SR(GIE);                             /* Disable Interrupts */
  initializeMCU();

  while (1)
  {
    if (P1IFG &= 0x02)                      /* If "state" btn pressed */
      {
      stateButtonPressed();
      stateTime = 0;                        /* Reset state timer */
      }
    else if (P1IFG &= 0x04)                 /* Or if "set" btn pressed */
      {
      setButtonPressed();
      stateTime = 0;                        /* Reset state timer */
      }
    else if (SecCycleFlag)                  /* Or if ~1 sec has passed */
      {
      /* If in time_state, ensure it doesn't stay in state for more than 10 seconds */
      if (TIME_state == state)
        {
        stateTime += 1;
        if (stateTime > 9)
          {
          state = TEMP_state;               /* Loop back to to TEMP state  */
          stateTime = 0;                    /* Reset state timer */
          }
        }
      }

   Tupdate_counter += 1;                /* Increment the temperature update counter */
    if (tempCheck_period == Tupdate_counter)/* Is it time to check again? */
      {
      ADC12CTL0 = ADC12ON;                  /* Enable ADC */
      ADC12CTL0 = ADC12CTL0|ENC|ADC12SC;    /* Enable conversion */
      ADC12CTL0 ^= 0x0001;                 /* Toggle off the ADC12SC bit to start converison*/
      while ((ADC12IFG&1) == 0);            /* When conversion complete on AD0 */
       Rtherm_elem = 39 - ((char)((int)ADC12MEM0*10) / 9);/* Calculate Rtherm table element */
                                            /* Formula is specific to thermistor used */

     if (Rtherm_elem > 30)                  /* In case where reading > than 40degC */
      Rtherm_elem = 30;                     /* Fix at meaximum allowable temp */

     currentTemp = Rtherm_array[Rtherm_elem];/* Add base temp value to index */
     Tupdate_counter = 0;                   /* Reset the temperature update counter */

     if (currentTemp<setTemp)
      P1OUT |= 0x01;                        /* Turn on LED */
     else
```

```
       P1OUT &= ~0x01;                       /* Turn off LED */
     }

    displayState();
    SecCycleFlag = 0;                        /* Clear the second flag */

   if (P1IFG == 0)                           /* If all KBI flags handled */
    {
    _BIS_SR(LPM3_bits + GIE);                /* Re-enable interrupts and go to sleep */
    } /* Else, MCU will run main loop again until all flags cleared */
   /* Zzzzzzzzzzzzz */
 }
}


void incMinute(void)
{
  minute += 1;
  if (minute >= 60)
    {
    minute = 0;
    hour += 1;
    if (hour >= 24)
           hour = 0;
    }
}

void displayInt(unsigned int number, unsigned char field )
{
  signed char buffer[6];
  char i = 5;
  buffer[i] = 0;
  /* For up to 5 digits : max value for int = 65535 */

  /* Use a "do/while" loop to take care of the number = 0 */
  do
    {
    buffer[--i] = (char)(number % 10) + '0';/* Convert least significant character of the int
into ascii (+'0' = 30hex) -> string buffer */
    number /= 10;                           /* Moves next digit to least significant position */
    } while (number != 0);

  if ((field < (5-i)) && (field != 0))      /* If field width is less than number of digits
only o/p field width of string */
  i = 5 - field;
 if (field > (5-i))          /* If field width is more than number of digits add leading zeros */
    {
    while ((5-i) < field)
    buffer[--i] = '0';                      /* Add leading zeros */
    }
  transmitSPI(&buffer[i]);
}

void transmitSPI (signed char *textPointer)
{
  while(*textPointer != 0)                  /* While not end of string */
    {
```

```
    UCA0CTL1 = 0x40;                          /* Release USCIA0 from reset state */
    UCA0TXBUF = *textPointer;                 /* Write the character to the USCIA0 interface */
    while ((UCA0STAT&UCBUSY) == 1);           /* Wait until transmission is complete */
    UCA0RXBUF;                                /* Read Rx Buffer to prevent SPI error */

    if (UCA0STAT != 0)                        /* If an error flag is set */
      {
      UCA0TXBUF = *textPointer;               /* Re-transmit the message */
      while ((UCA0STAT&UCBUSY) == 1);         /* Wait until re-transmission is complete */
      }

    UCA0CTL1 = 0x41;                          /* Place USCIA0 in reset state */
   textPointer += 1;                 /* Increment to point at the next character in the string */
    }
}

void displayState(void)
{
  switch (state)
    {
    case (TEMP_state):
      displayInt(currentTemp,sizeof(char)); /* Transmit Thermistor Read temperature */
     P1OUT = ((P1OUT&0x01)|(state<<6));    /* Display to user the current state being transmitted
without affecting Heat LED */
      break;
    case (SET_state):
      displayInt(setTemp,sizeof(char));     /* Transmit Target temperature */
     P1OUT = ((P1OUT&0x01)|(state<<6));    /* Display to user the current state being transmitted
without affecting Heat LED */
      break;
    case (TIME_state):
      displayInt(hour,sizeof(char));  /* Transmit the time (3 numbers: hours, minutes and
seconds) */
      displayInt(minute,sizeof(char));
      displayInt(second,sizeof(char));
     P1OUT = ((P1OUT&0x01)|(state<<6));    /* Display to user the current state being transmitted
without affecting Heat LED */
      break;
    }
}

void stateButtonPressed(void)
{
 if ((state == SET_state) && (*set_point != setTemp))  /* Set State and temp if different */
    *set_point = setTemp;                     /* Save new setpoint value */

  state += 1;                                 /* Increase State */

  if (state > TIME_state)                     /* If state has gone passed TIME */
    state = 0;                                /* Loop back around to TEMP */

  switchDelay();                             /* Short (200ms) delay to avoid switch debounce */
}

void setButtonPressed(void)
{
  if (state == SET_state)
```

```
    {
    setTemp += 1;                           /* Increase setpoint temp */
    if (setTemp > maximumT)                 /* Loop back if passed maximum */
      setTemp = minimumT;
    }

  else if (state == TIME_state)
    incMinute();

  switchDelay();                           /* Short (200ms) delay to avoid switch debounce */
}

void switchDelay(void)
{
  TACCTL0 = CCIE;                          /* Enable CCR0 interrupt */
  TACCR0 = 6550 - 1;                       /* Value reqiured for delay of ~200mS */
  TACTL = TASSEL_1+TACLR +MC_1;            /* ACLK=32kHz; TAR=0; up state and start Timer */
  _BIS_SR(LPM3_bits + GIE);                /* Go to sleep */
  /* Zzzzzzzzzzzzz */
  TACTL = 0;                               /* Disable timer */
  TACCR0 = 0;
  P1IFG = 0x00;                            /* Clear Flags */
  BtnHandlerFlag = 0;                      /* Allow RTC Interrupts */
}

/* Execute these functions immediately after reset, prior to variable initialization; a good
practice to ensure nothing pushes the watchdog stop instruction beyond the watchdog period */
void __low_level_init(void)
{
  int delayCycles;

  WDTCTL = WDTPW+WDTHOLD;                  /* Stop watchdog timer */
  FLL_CTL0 = XCAP18PF;                     /* Set load capacitance for xtal */

  /* wait for the crystal to stabilize.  Set up timer... */
  delayCycles = 150;                       /* Count to 0x0FFF 150 times */
  TACCR1 = 0x0FFF;
  TACCTL1 = CCIE;                          /* Compare state, interrupt */

  while ((IFG1&OFIFG) && (delayCycles > 0)) /* While osc in fault, count to */
    {                                      /* to 0x3FFF */
    delayCycles--;                         /* Decr the counter */
    IFG1 = 0;                              /* Clear OFIFG */
    TACTL = TASSEL_2 + TACLR + ID_3 + MC_2; /* SMCLK/8, TA clear, continuous state */
    _BIS_SR(LPM0_bits + GIE);              /* Now wait & see if it sets again */
    }

  /* If still not stable, something's wrong.  Handle it. */
  if (OFIFG&IFG1)
    _BIS_SR(LPM4_bits + GIE);              /* Handler if xtal never stabilizes */

  IFG2 = 0x00;                             /* Clear flag register 2 */
}

void initializeMCU (void)
{
  SCFQCTL = 127;                           /* set MCLK to max MHz with 32kHx crystal */
```

```
  SCFI0 |= FN_2;                            /* Seelect DCO Range */

  BTCTL = BT_ADLY_1000;                     /* ~1 second */
  IE2 = BTIE;                               /* Enable Basic Timer interrupts */

  /* Initialize SPI (using USCI_A0)  */
  UCA0CTL1 = 0x01;                          /* Hold in reset state */
 UCA0CTL0 = 0x6B;               /* Master - MSB first - 4pin idle high for slave transmission */
 UCA0CTL1 = 0x41;                 /* Select ACLK as USCI clock source and hold in reset state */
  UCA0BR0 = 0x01;
  UCA0BR1 = 0x00;                           /* Set SPI Bit Rate = Module Clock */
  IFG2 = 0x00;                              /* Clear all flags */
  P7SEL = 0x0F;                             /* Enable SPI pins */
  P7DIR = 0xFF;                             /* Set port to outputs */
  IE2 = IE2|UCA0TXIE;                       /* Enable Tx interrupt on USCIA0/SPI */

  /* Initialize I/O pins  */
  P1SEL = 0x20;                             /* Enable ACLK & MCLK out */
  P6SEL = 0x40;                             /* Enable AD0 */
  P1OUT = 0x00;                             /* All low */
  P1DIR = 0xF9;                             /* Outputs = all but SET and MODE functions */
  P1IES = 0x06;                             /* P1 edge select = high to low */
  P1IE  = 0x06;                             /* P1 interrupts enabled */
  P1IFG = 0x00;                             /* Clear port 1 flags */
  P2DIR = 0xFF;                             /* Set port to outputs */
  P3DIR = 0xFF;                             /* Set port to outputs */
  P4DIR = 0xFF;                             /* Set port to outputs */
  P5DIR = 0xFF;                             /* Set port to outputs */
  P6DIR = 0xFE;                             /* Set port to outputs (excl AD0/P6.0) */
  P8DIR = 0xFF;                             /* Set port to outputs */
  P9DIR = 0xFF;                             /* Set port to outputs */
  P10DIR = 0xFF;                            /* Set port to outputs */

  /* Configure the ADC  */
  ADC12CTL0 = 0x0000;                       /* 4 cycle sample, ADC12 diabled */
  ADC12CTL1 = 0x0010;                       /* Select MCLK as ADC12 clock source */
  ADC12MEM0 = 0x0000;                       /* Clear memory register */
  ADC12MCTL0 = 0x00;                        /* AVcc and AVss, Input Channel A0 */
  ADC12IE = 0x0000;                         /* Interrupt on AD0 disabled */

  /* Configure set point  */
  *set_point = setTemp;                     /* Init setpoint to 22 deg C */
}

/*****************************************************************************/

/* Timer A1 interrupt service routine */
#pragma vector=TIMERA1_VECTOR
__interrupt void Debounce(void)
{
   _BIC_SR_IRQ(LPM3_bits);                  /* Exit LPM3 on RETI */
   CCTL1 &= ~CCIFG;                         /* Clear CCR1 interrupt flag */
}

/* Basic timer ISR */
#pragma vector=BASICTIMER_VECTOR
__interrupt void Real_Time_Clock(void)
```

```
{
  second += 1;                              /* Add second */
  if (second >= 0x60)                       /* If minute have occured */
    {
    second = 0;                             /* Clear Seconds */
    incMinute();                            /* Add Minute */
    }

  SecCycleFlag = 1;                        /* Set flag to handle in main() next time it runs */
  if (!(BtnHandlerFlag))                   /* Don't interrupt debounce timer in the button
handlers */
    {
    P1IE &= ~0x06;                          /* Prevent port1 from disruptng */
    _BIC_SR_IRQ(LPM3_bits);                 /* Exit LPM3 */
    }
}

/* USCIA0/B0 Transmit ISR */
#pragma vector=USCIAB0TX_VECTOR
__interrupt void SPI(void)
{
  IFG2 = 0x00;                              /* Clear all SPI interrupt flags */
}
```

A.2 QE128 Code

```
/* Code by Inga Harris for MC9S08QE128  */
/* Freescale Semiconductor  */
/* September 2007  */
/* Built with CodeWarrior for Microcontrollers v6.0 */

#include <hidef.h> /* for EnableInterrupts macro */
#include "derivative.h" /* include peripheral declarations */

#pragma MESSAGE DISABLE C4000  /* Disable Condition Always True warning */
#pragma MESSAGE DISABLE C4002  /* Disable Result Not Used warning */

/* Define states */
#define TEMP_state         0               /* Temperature state */
#define SET_state          1               /* Set target temp/time state */
#define TIME_state         2               /* Clock state */

/* Constants */
#define minimumT          10               /* 10 deg C (~50 deg F) */
#define maximumT          40               /* 40 deg C (~100 deg F) */
#define tempCheck_period  30               /* Measure temp every 30 seconds */

/* Clock values */
unsigned char hour = 0x0C;                /* hours (tens/ones): starts from power on or reset
at midday */
unsigned char minute = 0x00;               /* minutes (tens/ones) */
unsigned char second = 0x00;               /* seconds (tens/ones) */

/* Temperature values */
unsigned char currentTemp;                 /* Initialise current temperature to 22degC */
unsigned char setTemp = 22;                /* Set Point temp */
char *set_point = (char *)0x00000080;      /* Set temperature pointer located in RAM @ 0x80 */
```

```
/* char *set_point = (char *)0x00800000; */ /* Set temperature pointer located in RAM @ 0x80_0000
*/

/* state controls */
unsigned char state = TEMP_state;           /* Specifies the user state */

/* Flags for communication between ISRs and main */
char BtnHandlerFlag = 0;                    /* Indicate btn press being handled */
char stateButtonFlag = 0;                   /* Run state rountine next loop itteration */
char SecCycleFlag = 0;                      /* Run 1sec cycle next loop itteration */
char SetButtonFlag = 0;                     /* Run set routine next loop itteration */

/* Temperature array */
const unsigned char Rtherm_array[31] = {10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
                                        20, 21, 22, 23, 24, 25, 26, 27, 28, 29,
                                        30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40};

/* Function declarations */
void configureADC(void);                    /* Routine to configure ADC */
void configureICS(void);                    /* Routine to configure ICS for full run state */
void configureSPI(void);                    /* Routine to configure SPI2 */
void displayState(void);                    /* Top level general display function */
void displayInt(unsigned int, unsigned char);/* Display numbers, includes ASCII conversion */
void enterLPR(void);       /* Handles ICS and SPMSC registers */
void enterStop3(void);
void incMinute(void);
void initializeMCU(void);
void stateButtonPressed(void);              /* Handles press of "state" switch */
void setButtonPressed(void);                /* Handles press of "set" switch */
void switchDelay(void);                    /* Short (200ms) delay to avoid switch debounce */
void transmitSPI (signed char *textPointer);/* Handles SPI transmission */

/*****************************************************************************/

void main (void)
{

  /* Set updateTemp_count close to tempCheck_period to minimize delay between "run" and display
*/
  unsigned int updateTemp_count = tempCheck_period - 1;       /* Cycles since temp last checked
*/
  unsigned int stateTime = 0;               /* Cycles spent in current state */
  unsigned char Rtherm_elem;                /* Temperature Table element number */

  /* INTC_WCR = 0x80; */                    /* Required for MCF51QE128 */
  DisableInterrupts                         /* Disable interrupts */
  initializeMCU();

  /* Main loop cycles forever */
  while (1)
  {
    if (KBI2SC_KBF && PTDD_PTDD4)           /* If "state" btn pressed */
    {
      stateButtonPressed();
      stateTime = 0;                        /* Clear state time count */
    }
    else if (KBI2SC_KBF && PTDD_PTDD5)      /* Or if "set" btn pressed */
```

```
    {
      setButtonPressed();
      stateTime = 0;
    }

    if (SecCycleFlag)                        /* If ~1 sec has passed */
    {
      /* If time_state, ensure it doesn't stayed too long (no more than 10 seconds) then loop
back to to TEMP state */
      if (state == TIME_state)
      {
        stateTime++;
        if (stateTime > 9)
        {
          state = TEMP_state;
          stateTime = 0;
        }
      }

      /* Check if it's time to get new temperature */
      updateTemp_count++;                    /* Cycles since last temp update */
      if (updateTemp_count == tempCheck_period) /* Time to check again? */
      {
         SCGC1_ADC = 1;                          /* Gate on ADC */
         configureADC();                         /* Re-initialize ADC */
         ADCSC1_ADCH = 0;  /* Start Conversion on Channel 0 */
         while (ADCSC1_COCO == 0);          /* When conversion complete on AD0 */
         SCGC1_ADC = 0;                           /* Gate off ADC */

         Rtherm_elem = 39 - ((char)((int)ADCR*10) / 9); /* Calculate Rtherm table element */
                                                 /* Formula is specific to thermistor used */

         if (Rtherm_elem > 30)
          Rtherm_elem = 30;

         currentTemp = Rtherm_array[Rtherm_elem];  /* Add base temp value to index */
         updateTemp_count = 0;

         if (currentTemp<setTemp)
          PTESET = 0x01;                        /* Turn on Heat LED */
         else
          PTECLR = 0x01;                         /* Turn off Heat LED */
       }
   }

   displayState();
   SecCycleFlag = 0;

   if (!KBI2SC_KBF)                             /* If all KBI flags handled */
   {
         EnableInterrupts;                /* Re-enable pin interrupts */
         enterStop3();           /* Go to sleep */
   } /* Else, MCU will run main loop again until all flags cleared */
  } /* Zzzzzzzzzzzzz */
}

/*****************************************************************************/
```

```
void initializeMCU (void)
{
  /* General System Control */
  SOPT1 = 0x23;    /* Enable Stop state and reset pin */
  SCGC1 = 0x00;               /* Gate off unused modules */
  SCGC2 = 0x94;               /* Only leave DBG, KBI and RTC on */

  configureICS();

  /* Configure RTC to interrupt ~1 second */
  RTCSC = 0x36;/* Enable interrupts and use OSCOUT to generate a 1 second period. */

  /* Initialize I/O pins (KBI initialisation included) */
  PTAD = 0x00;
  PTAPE = 0xFF;/* Pull-ups enabled */
  PTADD = 0x00;/* Input */
  PTBD = 0x00;
  PTBPE = 0xFF;/* Pull-ups enabled */
  PTBDD = 0x00;/* Input */
  PTCD = 0x00;
  PTCPE = 0xFF;/* Pull-ups enabled */
  PTCDD = 0x00;/* Input */
  PTDD = 0x00;
  PTDPE = 0xFF;/* Pull-ups enabled */
  PTDDD = 0x00;/* Input */
  KBI2SC = 0x04;/* KBI edge, interrupts disabled, clear any flag */
  KBI2PE = 0x30;/* Port D4 and D5 enabled */
  KBI2ES = 0x00;/* Falling Edge */
  PTED = 0x00;
  PTEPE = 0xFF;/* Pull-ups enabled */
  PTEDD = 0xC1;/* Input except PortE0 which is the SYSon LED */
  PTFD = 0x00;
  PTFPE = 0xFF;/* Pull-ups enabled */
  PTFDD = 0x00;/* Input */
  PTFD = 0x00;
  PTFPE = 0xFF;/* Pull-ups enabled */
  PTFDD = 0x00;/* Input */
  PTGD = 0x00;
  PTGPE = 0xFF;/* Pull-ups enabled */
  PTGDD = 0x00;/* Input */
  PTHD = 0x00;
  PTHPE = 0xFF;/* Pull-ups enabled */
  PTHDD = 0x00;/* Input */
  PTJD = 0x00;
  PTJPE = 0xFF;/* Pull-ups enabled */
  PTJDD = 0x00;/* Input */

  /* Configure setpoint */
  setTemp = *set_point;       /* Initialize setpoint */

}

void configureSPI (void)
{
  /* Initialize SPI (using SPI2) */
```

```
  SPI2C1 = 0x1E;            /* Master, Tx ISR enabled, MSB First, Idle Low, First Edge beginning
first cycle, SPI disabled */
  SPI2C2 = 0x10;            /* Normal state, Automatic SS */
  SPI2BR = 0x00;            /* Max Bus Clock */
  SPI2S;                   /* Read Status register to clear flag (dummy) */
  SPI2C1_SPE = 1;           /* Enable SPI 2 */
}

void configureADC (void)
{
  /* Configure the ADC */
  ADCSC1 = 0x1F;  /* Disable Interrupts and Module */
  ADCSC2 = 0x00;  /* Software trigger state, no compare */
  ADCCFG = 0x84;  /* Low Power configuration, 12-bit state using BusCLK/1 */
  APCTL1 = 0x01;  /* AD0 pin enabled */
  APCTL2 = 0x00;  /* All other ADC pins disabled */
  APCTL3 = 0x00;  /* All other ADC pins disabled */
}

void stateButtonPressed(void)
{
 if ((state == SET_state) && (*set_point != setTemp))  /* Set State and temp if different */
    *set_point = setTemp;                    /* Save new setpoint value */

  state += 1;                                /* Increase State */

  if (state > TIME_state)                    /* If state has gone passed TIME */
    state = 0;                               /* Loop back around to TEMP */

  switchDelay();                            /* Short (200ms) delay to avoid switch debounce */
}

void setButtonPressed(void)
{
  if (state == SET_state)
    {
    setTemp += 1;                            /* Increase setpoint temp */
    if (setTemp > maximumT)                  /* Loop back if passed maximum */
      setTemp = minimumT;
    }

  else if (state == TIME_state)
    incMinute();

  switchDelay();                            /* Short (200ms) delay to avoid switch debounce */
}

void switchDelay(void)
{
  SCGC1_TPM2 = 1;             /* Gate on TPM2 */
  /*  Delay to prevent re-run due to switch bounce */
  SPMSC2_LPWUI = 1;    /* Enable wake from wait to full run state */
  TPM2SC = 0x48;              /* Overflow interrupt enabled and Bus Clk as module source */
  TPM2MOD = 0x1900;    /* Mod of 6400 @ 32kHz is 200ms */
  enterLPR();
  _Wait;              /* Enter LPW */
```

```
 /* Zzzzzzzzzzzzz */

  TPM2SC = 0x00;              /* Disable timer */
  TPM2MOD = 0;        /* Clear Modulus */
  configureICS();             /* Get CPU back up to maximum speed */
  KBI2SC_KBACK = 1;           /* Clear KBI Flag */
  BtnHandlerFlag = 0;         /* Allow RTC_ISR to exit Stop3  */
  SCGC1_TPM2 = 0;             /* Gate off TPM2 */
}

void displayState(void)
{
  switch (state)
  {
    case (TIME_state):
      displayInt(hour,sizeof(char));  /* Displays the time (3 numbers: hours, minutes and
seconds) */
      displayInt(minute,sizeof(char));
      displayInt(second,sizeof(char));
      PTESET = 0x40;                        /* PTE7 on: PTE6 on */
      break;
    case (TEMP_state):
      displayInt(currentTemp,sizeof(char));
      PTESET = 0x40;
/* Display to user the current state being transmitted without affecting Heat LED */
      PTECLR = 0x80;
/* PTE7 off: PTE6 on (need to modify both to cover the 1st itteration where none are set) */
      break;
    case (SET_state):
      displayInt(setTemp,sizeof(char));
      PTESET = 0x80;                        /* PTE7 on */
      PTECLR = 0x40;                        /* PTE6 off */
      break;
  }
}

void displayInt(unsigned int number, unsigned char field )
{
  signed char buffer[6];
  char i = 5;
  buffer[i] = 0;
  /* For up to 5 digits : max value for word = 65535 */

  /* Use a "do/while" loop to take care of the number = 0 */
  do
    {
    buffer[--i] = (char)(number % 10) + '0';/* Convert least significant character of the word
into ascii (+'0' = 30hex) -> string buffer */
    number /= 10;                          /* Moves next digit to least significant position */
    } while (number != 0);

  if ((field < (5-i)) && (field != 0))      /* If field width is less than number of digits
only o/p field width of string */
  i = 5 - field;
  if (field > (5-i))                       /* If field width is more than number of digits add
leading zeros */
    {
```

```
    while ((5-i) < field)
    buffer[--i] = '0';                      /* Add leading zeros */
    }
  SCGC2_SPI2 = 1;                           /* Gate on SPI2 */
  configureSPI();                           /* Re-initialize the SPI2 registers */
  transmitSPI(&buffer[i]);
  SCGC2_SPI2 = 0;                 /* Gate off SPI2 */
}

void transmitSPI (signed char *textPointer)
{
        while(*textPointer != 0)                /* While not end of string */

        {
     SPI2C1_SPE = 1;           /* Enable SPI 2 */
     while (SPI2S_SPTEF == 0);/* Wait until Transmit buffer is empty */

     /* write the character to the SPI2 interface */
     SPI2D = *textPointer;
     while (SPI2S_SPTEF == 0);              /* Wait until transmission is complete */

     if (SPI2S_MODF != 0)                   /* If an error flag is set */
     {
        SPI2S_MODF;                         /* Clear SPI error flag */
        SPI2D = *textPointer;               /* Re-transmit the message */
        while (SPI2S_SPTEF == 0);           /* Wait until re-transmission is complete */
     }
     /* increment to point at the next character in the string */
     textPointer += 1;
     SPI2C1_SPE = 0;           /* Disable SPI2 */
        }
}

void enterLPR(void)
{
   ICSC1_IREFS = 0;/* select Ext ref clock */
   ICSC1_CLKS = 2;
   ICSC2_LP = 0;
   ICSC2_BDIV = 0;/* Run at 32kHz(CPU) - BDIV / 1 */
   ICSC2_LP = 1;           /* Enter FBELP */
   while (ICSSC_IREFST && (ICSSC_CLKST != 0x10));
      /* Wait for status bits to update then enter LPR */
   SPMSC1 ^= 0x0C;/* Clear LVDE ane LVDSE */
   SPMSC2_LPR = 1;/* Enter LPR state */
}

void configureICS(void)
  {
  /* FEE @ 50.33MHz */
  ICSC2 = 0x07;                         /* Low Range, Low Power, BDIV_1, Ext ref selected */
  ICSC1 = 0x00;                           /* FLL Enabled, int osc disabled */
  ICSSC_DRST_DRS = 0x2;
  ICSSC_DMX32 = 0;                        /* FLL factor of 1536 */
  while (ICSSC_IREFST == 1);              /* Wait for ext clock as ref indicator */
  }

void enterStop3(void)
```

```
{
   SPMSC1 ^= 0x0C;       /* Clear LVDE ane LVDSE */
   _Stop;
}

void incMinute(void)
{
  minute += 1;
  if (minute >= 60)
    {
    minute = 0;
    hour += 1;
    if (hour >= 24)
           hour = 0;
    }
}

/*****************************************************************************/

interrupt VectorNumber_Vtpm2ovf void _TPM2_OVF_ISR(void)
{
   TPM2SC;                                                                  /* Dummy
Read */
   TPM2SC_TOF = 0;    /* Clear Flag */
}
interrupt VectorNumber_Vspi2 void _SPI2_ISR(void)
{
if (SPI2S_SPRF)
   {
   SPI2S;         /* Dummy Read */
   SPI2D;         /* Dummy Read to clear flag */
   }

if (SPI2S_SPTEF);
/* If the Transmit Buffer Empty Flag set ignore and continue */

if (SPI2S_MODF)
   {
   SPI2S;         /* Dummy Read */
   displayState();/* Retramsit message */
   }
}
interrupt VectorNumber_Vrtc void _RTC_ISR(void)
{
  RTCSC_RTIF = 1;           /* Clear Flag */
/* Handle the clock tick in the ISR to ensure it happens immediately. */
   second++;
   if (second >= 0x60)
      {
      second = 0;
      incMinute();
      }

SecCycleFlag = 1;         /* Set flag to handle in main() next time it runs */
if (!(BtnHandlerFlag))    /* Don't interrupt debounce timer in the button handlers */
   KBI2SC_KBIE = 0;    /* Prevent KBI from disrupting */
}
```

THIS PAGE IS INTENTIONALLY BLANK

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor, Inc.
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
+1-800-521-6274 or +1-480-768-2130
www.freescale.com/support

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Document Number: AN3506
Rev. 0
12/2007

freescale™
semiconductor